システムLSI 検証技術入門

デバイスの記事

第13回

論理回路のプロパティ検証技術（5）

――プロパティ記述言語（PSL）の構文と使いかた

藤田昌宏

今回は，プロパティ（仕様）を記述する言語の一つであるPSLについて解説する．PSLはAccelleraが標準案をまとめ，現在，IEEEが仕様策定を進めている検証言語である．多くのEDAベンダがPSLのサポートを表明している．PSLは，コメント文としてHDLの設計記述に埋め込むなどして使用する．時相論理を記述できるほか，信号値のシーケンスをシンプルに表現するための記法などが定義されている．（編集部）

前回説明した時相論理や正規表現など，プロパティを記述するしくみを言語としてまとめたものの一つに「PSL（Property Specification Language）」があります．PSLは設計言語などの標準化・普及推進組織であるAccelleraが標準化案をまとめています．

標準化案となっている仕様記述手法は，PSL以外にもいくつかあります．言語によるもの，ライブラリ関数的な方法によるものなど，表現手法はいろいろありますが，基本的なしくみや考えかたは同じです．

ここでは，それらの代表としてPSL言語について説明します．また，前回までに説明した時相論理ではすべての動作を設計の状態ごとに表現する形になっていましたが，状態の集まりである「期間（interval）」ごとに記述するインターバル時相論理についても紹介します．

● PSLは四つの階層を定義

仕様記述言語の標準案とされているPSLは，米国IBM社の研究所で研究・開発されたSugarと呼ばれるものをもとに，さまざまな関係者の意見を取り入れてでき上がった言語です．Sugarを研究・開発していた研究者は，もともと時相論理を使ったハードウェアの仕様記述と形式的検証手法について研究してきた人たちなので，PSLの基本も時相論理であり，前回説明した正規表現なども含めて言語の形にまとまっています．

PSLでは，以下の四つの仕様表現の階層があります．

- **ブーリアン（boolean）層**
- **モデリング（modeling）層**
- **テンポラル（temporal）層**
- **検証（verification）層**

それぞれの階層について説明していきます．

1）ブーリアン層

もっとも基本となるのは，ブーリアン層と呼ばれる階層で，設計対象の状態を論理式の形で記述します．これは，各状態で成立すべきことを通常の論理式（時相演算子を含んでいないという意味）で表現します．一般に，通常の論理式には時間の概念がありません．例えばa＋b（信号aまたは信号bが'1'であるという論理式）と書くと，a＋bがすべての状態（つまり，すべての時刻）で成立することになってしまいます．一方，これを時相論理式として解釈すると，「現在の時刻でa＋bが成立する」ということのみを意味します．すなわち，時相論理式として記述されたものに時相演算子が現れないと，その時刻やその状態でのみ成立する式であると解釈されるわけです．したがって，ブーリアン層では各状態で成立すべきことを状態に合わせて論理式で表現することになります．

2）テンポラル層

次がテンポラル層と呼ばれる階層です．この階層では設計対象の時間軸上における動作シーケンスが記述されます．記述には，基本的に時相論理や正規表現などが利用されます．この部分は，前回までに説明してきたプロパティの記述法がそのまま当てはめられます．時相論理や正規表現を使うことでかなり柔軟にプロパティを表現できますが，それでもいくつかのプロパティについては，表現が複雑になったり，場合によっては表現できないケースもありました．

例えば，偶数時刻でのみ信号aが‘1’となるが，奇数時刻では信号aに関する条件はない（つまり，値は‘0’でも‘1’でもよい）というプロパティは，時相論理では直接記述することができません．しかし，状態遷移図を利用すると，この性質は**図1**に示すように簡単に表現できます．

3）モデリング層

そこで，次の階層としてモデリング層と呼ばれるものが定義されています．モデリング層では必要に応じて中間変数なども導入して，状態遷移の形で動作を表現します．設計が想定しているまわりの動作環境などの表現としても利用できます．中間変数を利用して状態遷移を一般的に表現できることから，実質的に，任意の複雑度のプロパティを（言い換えると，一種の設計記述も）表現できます．また，かなり複雑なインターフェース条件も記述可能です．

4）検証層

最上位の階層は検証層と呼ばれています．この階層では，検証ツールに対して，記述されているプロパティが成立することを要求するのか，それとも記述されているプロパティが成立すると仮定して設計対象を検証することを要求するのかなどを指定します．表現されているプロパティは設計が満たすべき仕様を示しているのですが，設計はある種の動作環境における動作のみを保証すればよく，任意の入力シーケンスを考える必要がない場合が多々あります．このような場合，設計が「仮定」している動作環境をプロパティに含める必要があります．検証層ではassumeというキーワードを使って，これを表現できるようになっています．

● コメント文としてHDL記述に埋め込んで使う

PSLの記述例を見ていきましょう．実際には，PSLは既存のハードウェア記述言語であるVHDLやVerilog HDLを拡張する形で定義されています．以下は，VHDLを拡張した形のPSLについて説明していきます．なお，Verilog HDLの場合も，構文がVerilog HDL用に変わるだけなので，内容はすべて同じと考えて差し支えありません．

PSLの記述例として，まず，次の記述を考えます．

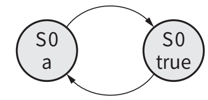


図1　時相論理で記述できないプロパティの例
偶数時刻でのみ信号aが‘1’となるが，奇数時刻では信号aに関する条件はない（つまり，値は‘0’でも‘1’でもよい）というプロパティは，時相論理では直接記述することができない．しかし，状態遷移図を使うと，この性質を簡単に表現できる．

```
-- psl property FIFO_P1 is always
((not push) or pop or (not full))
                    @rising_edge(CLK1);
```

この記述は，**図2**に示すように分解されて解釈されます．まず，--はVHDLではコメントなので，VHDLの通常の処理系から見るとPSL記述はすべてコメントということになります．-- pslと記述すると，PSLの処理系（PSL記述を理解できるシミュレータや検証ツール）は，それ以降の文をPSLの記述と認識します．

次のキーワードpropertyは，以降がプロパティの記述であることを示し，次にプロパティ名（この場合はFIFO_P1）が続きます．そして，次のキーワードis以下がプロパティFIFO_P1の定義となります．次のalwaysで以下のプロパティは「いつも成り立つべきである」ということを表現しています．その次に，VHDLの構文に沿った形で論理式で条件が示されています．この場合，FIFOの操作と状態は，「push操作を行っていないか，pop操作を行っているか，fullではないかのいずれかの場合である」ということが表現されています．これは一見わかりにくい論理式ですが，じつはこの否定の論理式は以下のようになります．

(not ((not push) or pop or (not full)))
= ((not (not push)) and (not pop) and (not (not full)))
= (push and (not pop) and full)

否定の論理式に対応するFIFOの操作は，fullの状態のときにpopせずにpushすることになり，オーバフローしてしまうことになります．したがって，このようなことは

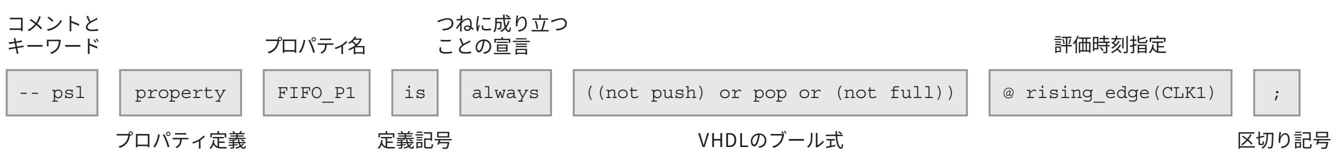


図2　PSLの記述例
ここではVHDLを拡張した形のPSLの例を示している．図のように分解されて解釈される．

つねに起きてはいけないということがPSLで記述されているわけです．

次の@ rising_edge(CLK1)は，以上の条件をチェックする時刻がクロックCLK1の立ち上がりエッジごとであることを示しています．PSLでは，このような形でいろいろなプロパティを，必要に応じて，VHDLやVerilog HDLの設計記述内に挿入していくことになります．このプロパティでは，条件は時相演算子のない通常の論理式((not push) or pop or (not full))ですが，一般には，条件を時相論理式や正規表現を利用して記述することもできます．例えば正規表現を利用したプロパティとして，次のようなものを記述できます．

```
{true[*];req;ack} |=> {start;busy[*];end}
```

|=>は，その左辺が成立すると右辺が成立しなければならないことを表現しています．時間軸上の信号値のシーケンスは { と } でくくって表現します．つまり，この式は{true[*];req;ack}というシーケンスが起きたときに，{start;busy[*];end} というシーケンスが成り立たないといけないという意味です．{ }内では，時間の区切りは ; で表現します．ですからreq;ackというのは，最初の時刻で信号reqが‘1’になり，次の時刻で信号ackが‘1’になるという意味です．また，ある時刻で信号値に対する制限がいっさいないことはtrueで表現します．同じことがn回繰り返されることは，その信号の後に[n]と記述して表現します．その回数が不定の場合は[*]と記述します．

したがって，{true[*];req;ack}は，まず何も起こらないことが一定期間続き，その後，信号reqが‘1’になり，その次の時刻で信号ackが‘1’になるというシーケンスを示しています．また，{start;busy[*];end}は，最初の時刻で信号startが‘1’になり，次の時刻で信号busyが‘1’になることが一定期間続き，その後の時刻で信号endが‘1’になるシーケンスを示しています．したがって，{true[*];req;ack} |=> {start;busy[*];end} 全体では，reqが‘1’になり，その次にackが‘1’になると，startが‘1’になり，その次の時刻から一定期間busyが‘1’になります．つまり，ここで何らかの動作が行われてbusyになっていることを意味します．そして，busyが‘1’でなくなると，endが‘1’になることで処理が終了する，という動作を表現していることになります．

● 信号波形でプロパティの真偽を考える

具体的な信号値のシーケンスの例でこの式の真偽を表したものを**図3(a)**に示します．

信号trueは条件がないことを示すためのもので，いつも‘1’となっている信号であると便宜的に考えることができます．**図3(a)**ではreq，ack，start，busy，endの信号値の例が示されています．「左辺が成立」と書いた期間では，左辺の{true[*];req;ack}が成立しており，また，「右辺が成立」と書いた期間では，{start;busy[*];end}が成立しています．したがって，左辺が成立した期間の直後に右辺が成立する期間がかならずあるので，**図3(a)**のタイミング・チャートで表現された信号シーケンスは{true[*];req;ack} |=> {start;busy[*];end}をいつも満たしていると判定されます．

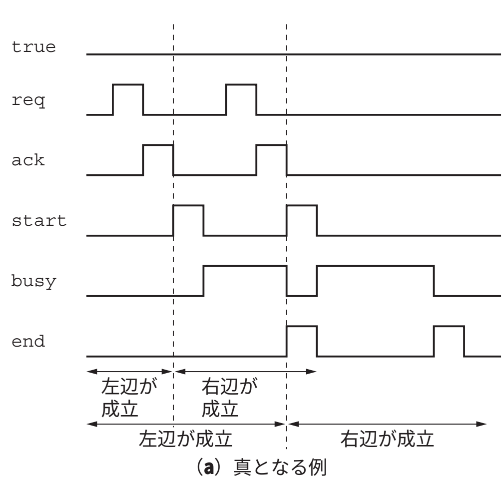


図3 {true[*];req;ack} |=> {start;busy[*];end}の信号波形
(b)では，信号busyの動作が一部おかしく，反応が1サイクル遅い箇所がある．

次に，**図3(b)**に示すタイミング・チャートを見てください．**図3(a)**との違いは，信号busyの動作が一部おかしく，反応が1サイクル遅い部分(「異常」となっている部分)がある点だけです．この部分では，信号req，ackは左辺を成立させているにもかかわらず，信号busyの反応が1サイクル遅いため，右辺は成立しません．したがって，この部分では {true[*];req;ack}|=>{start;busy[*];end} が成立しないことになります．検証ツールでこのことをチェックした場合，プロパティは成立せず，**図3(b)**の「異常」となっている部分で成立しないというメッセージや表示が示されることになります．

● PSLの基本構文を理解する

以上，PSL記述の例をいくつか示し，基本的な記述法とどのようなことがチェックされるのかについて簡単に説明しました．PSLにはプロパティを記述するための便利な機能が数多くあります．以下ではそれらについて紹介していきたいと思います．

まず，全体の文法をもう少し詳しく見ていきましょう．全体の構成は**リスト1**のようになります．なお，これはVHDLをベースとした場合です．Verilog HDLをベースとする場合にはisが=に変わるなど，Verilog HDL言語の構文に合わせていくつか変化します(基本は同じである)．

まず，-- psl propertyと記述して，PSLでプロパティの指定を行うことを宣言します．その後の<name>は，そのプロパティに付ける名まえです．そしてis以下がプロパティの定義になります．

[operator]には，alwaysやneverがあります．neverはalwaysの反対で，それ以降の記述の内容が決して起こらないことを宣言します．一般にプロパティは「AならばB」の形で記述され，Aが成立した場合にBが成立しなければならない，という形で表現されます．[enabling_condition]はAの部分に相当し，プロパティが成立すべき条件を指定する式が示されます．

また，次の次にある(fulfilling_condition)はBの部分に相当し，条件が満たされたときに成立すべき条件です．その間にある[implication_operator]は，Aの条件が成り立つ時刻とBが満たされるべき時刻の関係を指定する演算子で，->，->next，|->，|=>，eventually!の5種類があります．

->は左辺の条件が満たされるとすぐに右辺の条件をチェックすることを意味します．この場合，左辺は1時刻内の条件で，シーケンスは許されません．また->nextは，左辺が成立してから1サイクル後に右辺の条件がチェックされます．この場合も左辺は1時刻内の条件です．

これらに対し，|->は，左辺のシーケンスが成立した場合に，そのシーケンスの最後の時刻を最初の時刻とするシーケンスに対して右辺の条件が成立するかどうかがチェックされます．|=>は，左辺のシーケンスが成立した場合に，そのシーケンスの次の時刻から始まるシーケンスに対して，右辺が成立するかどうかがチェックされます．したがって，|=>は，|->{true;と書いたものと同じことになります．

最後のeventually!は，左辺のシーケンスが成立した場合に，いつかは右辺が成立するかどうかがチェックされます．シミュレーションでチェックしている場合には，シミュレーションが終了するまでに成立するかどうかがチェックされます．

このようにPSLでは，[implication_operator]を5種類用意しておくことにより，プロパティが成立すべき場合を指定する条件(左辺)とそれが満たされた場合に要求される条件(右辺)の時間的な関係を柔軟に表現できるようになっています．

次の[until | until_ | abort discharging_condition]は，成立すべき条件のチェックをいつまで行うかを示しています．untilの場合には，discharging_conditionが成立する時刻まで条件のチェックを続けることになります．また，abortの場合には，discharging_conditionが成立したら，処理の途中でもそのプロパティのチェックを中止することを意味しています．なお，これらの条件がない場合には，成立すべき条件のチェックは原理的に無限に続くことになります．つまり，シミュレーションでプロパティをチェックしている場合はそのシミュレーションが終了するまで，また，プロパティ検証ツール(property checker)でチェックしている場

リスト1　PSLの基本構文

```
-- psl property
      <name>
      is
      [operator]
      [enabling_condition]
      [implication_operator]
      (fulfilling_condition)
      [until | until_ | abort discharging_condition]
      [@(clock_expression)];
```

合は検証対象の動作がループになって同じ動作を繰り返すことが確認されるまで，実行されます．

最後の [@(clock_expression)] は，どのタイミングで各種条件をチェックするかを指定します．通常，検証対象は同期回路なので，クロック信号名と，立ち上がりか立ち下がりかの指定を記述します．例えばVHDLでは，clk'event and clk='1' のようになります．

● 信号値のシーケンスと繰り返しをシンプルに記述

次に各種条件を指定する際に利用する信号値のシーケンスの表現方法について，少し詳しく見ていきましょう．信号値のシーケンスはかっこ { } でくくって表現します．また，; で時間軸上のシーケンスの連結を表現します．これは，一種の時相演算子であると考えられます．

このようなシーケンスの連結をベースとした時相論理はインターバル時相論理と呼ばれています．インターバル時相論理は通常の時相論理より柔軟なプロパティ記述であり，さらに言うとハードウェアの動作記述も可能です．そのため，インターバル時相論理を基礎とするハードウェア記述言語やプログラミング言語が研究・開発されており，無償公開されたツールも存在します．このあたりについては別の機会に詳しく説明することにしたいと思います．

さて，この ; を使うことで，例えば {a; b; c} と記述すると，信号aが今の時刻 t で'1'となり，信号bが次の時刻 $t+1$ で'1'となり，信号cがその次の時刻 $t+2$ で'1'となるということを表しています．次に，ある信号値の繰り返しは[* と] を使って表現します．

[*]と書くと，その直前の信号値が任意の回数繰り返されるということになります．この記述は条件式の中で使用され，0回でも1回でも多数回でも，どれでも条件が満たされたと解釈されます．

これに対して，[n]（ただしnは整数値）と書くと，きっかりn回繰り返すという条件になります．また，繰り返し回数に下限と上限がある場合には，例えば [*n to m]（n，mは整数値でn＜m）と書くと，最小n回，最大m回の繰り返しを表現します．これを使うと，例えば「リクエスト信号が来てから4サイクル以内にアクノリッジ信号を返さなければならない」というプロパティ REQ_ACK_IN_4_CYCLE は，次のように記述できます．

```
-- psl property REQ_ACK_IN_4_CYCLES is
always({req and !ack} |=>{true[*0 to 3];
ack} abort !reset_n) @(clk'event and
                                 clk='1');
```

まず，{req and !ack}でリクエスト信号reqが'1'になり，かつそのときにアクノリッジ信号ackが'1'でなければ，その次の時刻から始まるシーケンスで（|=> なので次の時刻からになる），{true[*0 to 3]; ack} abort !reset_n が成立しなければならないことになります．true[*0 to 3] は何も条件がない状態（trueがこれを意味する）が0時間以上で最大3時間あることを表現します．また，その次の時刻で信号ackが'1'になること，つまり，左辺の {req and !ack} が成立した次の時刻から4時間以内にackが'1'にならなければならないことを意味しています．これはREQ_ACK_IN_4_CYCLEで表現すべきプロパティを正しく示していると言えます．

なお，abort !reset_nの部分は，4サイクル以内に応答しなければならないのはリセット信号が来ない場合であることを意味しています．このように，PSLではリセット条件やトラップ条件も自由に追加できるようになっています．

表1に，シーケンス作成のための繰り返し表現をまとめたものを示します．[*]は0回以上の繰り返しでしたが，[+]は1回以上の繰り返しを表します．また，infは予約語で無限大を表し，[*n to inf]はn回以上，任意回数の繰り返しを表現します．また，[*0 to m] とすると，0回以上，最大m回の繰り返しを表現することになり，ここで0回の繰り返しとは，その動作をスキップすることを意味します．最後の[=] は連続しない繰り返しの場合に利用します．これらの繰り返し表現を使って，プロパティを作った例を**表2**に示します．

表1　シーケンス記述のための繰り返し表現

記　述	意　味
[*n]	n サイクルの繰り返し
[*]	0回以上，任意回数の繰り返し
[+]	1回以上，任意回数の繰り返し
[*n to m]	最小 n 回，最大 m 回の繰り返し
[*n to inf]	n 回以上，任意回数の繰り返し
[*0 to m]	0回，つまりこの条件をスキップするか，または最大 m 回の繰り返し
[=]	連続しない繰り返しの場合の表現

● 部分シーケンスに名まえを付けて再利用

さて，シーケンスは ; 演算子を繰り返し表現といっしょ

表2 繰り返し表現によるプロパティの例

<table>
<tr><th>プロパティの例</th><th>意　味</th></tr>
<tr><td>always ({a} |=> {b[*2]; c});</td><td>aが‘1’になったら，b; b; cというシーケンスが発生する</td></tr>
<tr><td>always ({a} |=> {true[*2]; c});</td><td>aが‘1’になったら，-; -; c というシーケンスが発生する
（ただし，-は条件がないことを表すものとする）</td></tr>
<tr><td>always ({a} |=> {b[*1 to 3]; c});</td><td>aが‘1’になったら，b; cまたはb; b; cまたはb; b; b; cのシーケンスが発生する</td></tr>
<tr><td>always ({a} |=> {b[*0 to 3]; c});</td><td>aが‘1’になったら，cまたはb; cまたはb; b; cまたはb; b; b; cのシーケンスが発生する</td></tr>
<tr><td>always ({a} |=> {b[+]; c});</td><td>aが‘1’になったら，b; cまたはb; b; cまたはb; b; … ; b; cのシーケンスが発生する</td></tr>
<tr><td>always ({a} |=> {[*]; b; c});</td><td>aが‘1’になったら，b; cというシーケンスが発生してこのプロパティが終了する．また，b; cというシーケンスが発生しなくてもかまわない</td></tr>
<tr><td>always ({a} |=> {b[=2]; c});</td><td>aが‘1’になったら，cが‘1’になるまでにbが2回‘1’になる</td></tr>
</table>

に使って柔軟に記述できますが，さらに表現しやすくするために，部分シーケンスに名まえを付けて再利用できるようになっています．記述が短くなるだけでなく，部分シーケンスに名まえを付けることでプロパティが示している動作をユーザが理解しやすくなるという利点もあります．

シーケンスに名まえを付けて定義する際には-- psl sequenceを使います．例えば，以下の記述を考えます．

```
-- psl sequence AB is {a; b};
-- psl sequence CD is {c; d};
-- psl sequence AB_EV is {a; [*]; b};
```

この記述では{a; b}というシーケンスにABという名まえが，{c; d}というシーケンスにCDという名まえが付きます．また，aが‘1’になって，その後bが‘1’になるという{a; [*]; b}というシーケンスには，AB_EVという名まえが付けられます．そして，以下のようにいったん名まえを付けたシーケンスを使って，新たなシーケンスを定義して，名まえを付けることも可能です．

```
-- psl sequence EMBEDDED is {AB; CD};
```

また，プロパティもこれらのシーケンスの名まえを使って定義できます．例えば，以下のような記述です．

```
-- psl property DISJUCTION_TEST is always
              ({go} |=> {AB} | {CD});
```

この記述では，信号goが‘1’になると，シーケンスAB（つまり，{a; b}），またはシーケンスCD（つまり，{c; d}）のいずれかが起こることを表しています．また，以下のように名まえを付けたシーケンスの繰り返しを指定することもできます．

```
-- psl property FUSION_TEST is always
       ({go} |=> {AB[*2] }to {CD[*2] ]);
```

さらにPSLには，記述をわかりやすくするためのくふうとして，デフォルトのクロックを指定する表現も用意されています．

```
-- psl default clock is (clk'event and
                            clk = '0');
```

これにより，各プロパティでクロックを明示しなくてもよくなります．

このようにPSLを利用するとかなり複雑，かつ細かい動作を仕様として表現することができます．同様のことはVHDLやVerilog HDLを利用しても表現できなくはありません．しかし，実際に記述してみると，かなりたいへんです．例えば，以下のようなプロパティをVHDLやVerilog HDLなどで記述してみると，約50行程度になってしまいます．これが，PSLだと1行になるわけです．

```
always {[*];{request and !acknowledge}
                     [*5]}|->{busy_flag};
```

ふじた・まさひろ
東京大学 大規模集積システム設計教育研究センター（VDEC）教授

<筆者プロフィール>
藤田昌宏．1985年，東京大学大学院情報工学専門課程修了．工学博士．同年，富士通研究所入社．論理設計用CAD技術の研究開発に従事．1993年より米国富士通研究所へ出向，CAD研究グループを立ち上げる．2000年3月より東京大学大学院工学系研究科電子工学専攻教授，2004年4月より東京大学大規模集積システム設計教育センター（VDEC）教授．ディジタル・システム設計支援技術，特にシステム・レベル合成，論理合成，論理検証を中心に研究している．